Chapter 3 – Implementing Classes

Chapter Goals
- To become familiar with the process of implementing classes
- To be able to implement simple methods
- To understand the purpose and use of constructors
- To understand how to access instance fields and local variables
- To appreciate the importance of documentation comments

Black Box
- A black box is something that performs a task, but hides how it does it
- This is also called encapsulation
- Black boxes in a car: transmission, electronic control module, etc

Figure 1
Levels of Abstraction in Automotive Design

Levels of abstraction: Software Design

3.2 Designing the Public Interface to a Class
- Your task: Develop a BankAccount class
- First step: Define essential features
- Behavior of bank account (abstraction):
  - deposit money
  - withdraw money
  - get balance

Old times: computer programs only manipulated primitive types such as numbers and characters
Gradually programs became more complex, vastly increasing the amount of detail a programmer had to remember and maintain
Designing #1: Methods
- Methods of BankAccount class:
  - deposit
  - withdraw
  - getBalance
- Support method calls such as the following:
  - harrysChecking.deposit(2000);
  - harrysChecking.withdraw(500);
  - System.out.println(harrysChecking.getBalance());
- Which methods are accessors? Mutators?

Examples:
- public void deposit(double amount) {...}
- public void withdraw(double amount) {...}
- public double getBalance() {...}

Designing: Method Definitions
COMPONENTS:
- access specifier (Ex. public)
- return type (Ex. String or void)
- method name (Ex. deposit)
- list of parameters (Ex. amount for deposit)
- method body in braces {...}

Syntax 3.1: Method Definition
accessSpecifier returnType
   methodName(paramaterType parameterName, ...)
   {
     method body
   }
Example:
   public void deposit(double amount){
     ... 
   }
Purpose:
- To define the behavior of a method

Designing #2: Constructor
- A constructor initializes the instance variables
- Constructor name = class name

public BankAccount()
{
  // body--filled in later
}

Constructors
- Constructor body is executed when new object is created
- Statements in constructor body will set the internal data of the object that is being constructed
- How does the compile know which constructor to call?
Constructor vs. Method

- Constructors are a specialization of methods
  - Goal: to set the internal data of the object

- 2 differences
  - All constructors are named after the class
  - Therefore all constructors of a class have the same name
  - No return type listed EVER!

BankAccount Public Interface

- The public constructors and methods of a class form the public interface of the class.

```java
public class BankAccount {
    // private fields—filled in later

    // Constructors
    public BankAccount() {
        // body—filled in later
    }
}
```

3.3 Commenting the Public Interface

- Part of creating a well-defined public interface is always commenting the class and methods behaviors

- The HTML pages from the API are created from special comments in your program called javadoc comments

- Place before the class or method
  - /**
  - ..
  - */
Javadoc Comments
- Begin with "/**"
- Ends with "*/"
- Put "" on lines in between as convention, makes it easier to read
- **Javadoc tags** - @ mark is one
  - @author, @param, @return
- **Benefits**
  - Online documentation (Assignment 1)
  - Other java documents

/*
 * Withdraws money from the bank account.
 * @param the amount to withdraw
 */
public void withdraw(double amount){
    // implementation filled in later
}

/**
 * Gets the current balance of the bank
 * account.
 * @return the current balance
 */
public double getBalance(){
    // implementation filled in later
}

3.4 Instance Fields
- Remember: methods are the public interface of a class
- Instance Fields are part of the internal workings - An object stores its data in
  **instance fields**
  - Field: a technical term for a storage location inside a block of memory
  - **Instance of a class**: an object of the class
  - AKA Data Members

The class declaration specifies the instance fields:
```java
public class BankAccount{
    private double balance;
    ...}
```
An instance field declaration consists of the following parts:
- access specifier (usually private)
- type of variable (such as double)
- name of variable (such as balance)

Each object of a class has its own set of instance fields

You should declare all instance fields as private

Access Specifiers
- **Access Specifiers** – defines the accessibility of the instance field and methods
  - **private** – only accessible within the class methods
  - **public** – accessible in outside class methods and inside class methods

Private enforces encapsulation/black box

AKA Visibility Modifier

Syntax 3.4: Instance Field Declaration

```java
accessSpecifier class ClassName
{
    . . .
    accessSpecifier fieldType fieldName;
    . . .
}
```

Example:
```
public class BankAccount
{
    . . .
    private double balance;
    . . .
}
```

Purpose: To define a field that is present in every object of a class

Accessing Instance Fields

The deposit method of the BankAccount class can access the private instance field:
```
public void deposit(double amount)
{
    double newBalance = balance + amount;
    balance = newBalance;
}
```

Other methods cannot:
```
public class BankRobber
{
    public static void main(String[] args)
    {
        BankAccount momsSavings = new BankAccount(1000);
        . . .
        momsSavings.balance = -1000; // ERROR
    }
}
```

Encapsulation = Hiding data and providing access through methods

By making data members private, we hide internal workings of a class from a user

Note: We can have public instance fields and private methods, but commonly we do not
3.5 Implementing Constructors & Methods

- Constructors contain instructions to initialize the instance fields of an object.
  ```java
  public BankAccount()
  {
    balance = 0;
  }
  public BankAccount(double initialBalance)
  {
    balance = initialBalance;
  }
  ```

**Why put instructions in a method?**

... or why not just have one long list of instructions in our programs?

1. smaller ↔ easier
2. test & debug once, execute as often as needed
3. more readable code

**Constructor Call Example**

- `BankAccount harrysChecking = new BankAccount(1000);`
- Create a new object of type BankAccount
- Call the second constructor (since a construction parameter is supplied)
- Set the parameter variable initialBalance to 1000
- Set the balance instance field of the newly created object to initialBalance
- Return an object reference, that is, the memory location of the object, as the value of the new expression
- Store that object reference in the harrysChecking variable

**Implementing Methods**

- Some methods do not return a value
  ```java
  public void withdraw(double amount)
  {
    double newBalance = balance - amount;
    balance = newBalance;
  }
  ```

- Some methods return an output value
  ```java
  public double getBalance()
  {
    return balance;
  }
  ```

**Method Call Example**

- `harrysChecking.deposit(500);`
- Set the parameter variable amount to 500
- Fetch the `balance` field of the object whose location is stored in `harrysChecking`
- Add the value of amount to `balance` and store the result in the variable `newBalance`
- Store the value of `newBalance` in the `balance` instance field, overwriting the old value

**Syntax 3.5: The return Statement**

- `return expression;`
- `or`
- `return;`

**Example:**

```java
return balance;
```

**Purpose:**

- To specify the value that a method returns, and exit the method immediately. The return value becomes the value of the method call expression.
/** Stores information about one pet. */
public class Pet {
    private String name, kind;
    private int age;

    /** Initializes a new instance. */
    public Pet (String n, String k, int a) {
        name = n; kind = k; age = a;
    }

    /** Returns the name of this pet. */
    public String getName() { return name; }

    /** Changes the name of this pet. */
    public void changeNameTo(String newName) {
        name = newName;
    }
}

/** Stores information about one pet. */
public class Pet {
    private String name, kind;
    private int age;

    /** Initializes a new instance. */
    public Pet (String n, String k, int a) {
        name = n; kind = k; age = a;
    }

    /** Returns the name of this pet. */
    public String getName() { return name; }

    /** Changes the name of this pet. */
    public void changeNameTo(String newName) {
        name = newName;
    }
}

/** Deposits money into the bank account. */
public void deposit(double amount) {
    double newBalance = balance + amount;
    balance = newBalance;
}

/** Withdraws money from the bank account. */
public void withdraw(double amount) {
    double newBalance = balance - amount;
    balance = newBalance;
}

3.6 Testing Classes

- Previous section was designing a class
- By itself, we cannot actually run a program
  - No main() method, like most classes
- We create instances of it in other classes
- Idea: use a test class to make sure it works properly before dispensing the product
3.6 Testing A Class
Test class: a class with a main method that contains statements to test another class.

Typically carries out the following steps:
- Construct one or more objects of the class that is being tested
- Invoke one or more methods
- Print out one or more results
- Verify output is correct and program behaves as expected

```java
/**
 * A class to test the BankAccount class.
 */
public class BankAccountTester{
    /**
     * Tests the methods of the BankAccount class.
     * @param args not used
     */
    public static void main(String[] args){
        BankAccount harrysChecking = new BankAccount();
        harrysChecking.deposit(2000);
        harrysChecking.withdraw(500);
        System.out.println(harrysChecking.getBalance());
    }
}
```

3.7 Categories of Variables
Categories of variables
- Instance fields (balance in BankAccount)
- Local variables (newBalance in deposit method)
- Parameter variables (amount in deposit method)

- Share the same properties of declaring and creating
- Differ in their lifetime (AKA scope)

Instance Field
- An instance field belongs to an object
  - AKA Instance variable
  - Each object has its own copy of the instance field

- The fields stay alive until no method uses the object any longer

- More specifically, until the object no longer exists

Local Variables
- Local and parameter variables belong to a method
  - You declare them and create within the method

- When method is done executing, variable is thrown out
  - Every time the method is executed, a new copy of any variables is created, values do not carry over
**Lifetime Of Variables**

Say we made the following method call

```java
harrysChecking.deposit(500);
```

Remember that `deposit` looks like this

```java
public void deposit(double amount) {
    double newBalance = balance + amount;
    balance = newBalance;
}
```

**3.8 Implicit and Explicit Method Parameters**

- The implicit parameter of a method is the object on which the method is invoked.
- Why is this important?
- When a method is invoked, and an instance field is used, how does it know which object it belongs to?

- *balance* is the balance of the object to the left of the dot:

  ```java
  momsSavings.withdraw(500);
  ```

  means

  ```java
  double newBalance = momsSavings.balance - amount;
  momsSavings.balance = newBalance;
  ```

**Implicit Parameters and this**

- Every method has one implicit parameter.
- The method knows what object called it based on a reference, but does not know/care about the identifier name.
  - E.g., you cannot say `momsSavings.balance` in the class, because `momsSavings` is a local variable of another class.
- So in order to be more general, we call the implicit parameter *this*.
- Exception: Static methods do not have an implicit parameter (more on Chapter 9).
Implicit Parameters and this

- double newBalance = balance + amount;
  // actually means
double newBalance = this.balance + amount;

- When you refer to an instance field in a
  method, the compiler automatically applies
  it to the this parameter

Example – without this

```java
public class BankAccount{
    double balance;
    public BankAccount(double initialBalance){
        balance = initialBalance;
    }
    public BankAccount(){
        balance = 0;
    }
}
```

Example – with this

```java
public class BankAccount{
    double balance;
    public BankAccount(double initialBalance){
        balance = initialBalance;
    }
    public BankAccount(){
        this(0);
    }
}
```

HOW TO 3.1 – Designing and Implementing a Class

- Step 1: Find out what you are asked to do
  with an object of the class
  - What do you want to be able to do with the
    object?
- Step 2: Specify the Public Interface
  - Convert the list from step 1 into methods, and
    the parameters (think: input) they need
  - Constructors: how do I want to initialize this
    object

- Go back to Step 2 if implementation
  doesn’t work
- Step 6: Test your class

HOW-TO 3.1 (cont)

- Step 3: Document the public interface
  - Create Javadoc comments for each method
    and the class
- Step 4: Determine instance fields
  - What information needs to be maintained?
- Step 5: Implement
  - One at a time, from easiest to most difficult
Note on Style #1

- A couple ways to use curly braces

Option #1
public class SomeClass{
    
}

Option #2
public class SomeClass{
    
}

Note on Style #2

- Book says to place instance fields at the bottom of a class
public class SomeClass{
    //constructors
    //methods
    //instance fields
}

- Most conventions have it at the top of a class
public class SomeClass{
    //instance fields
    //constructors
    //methods
}